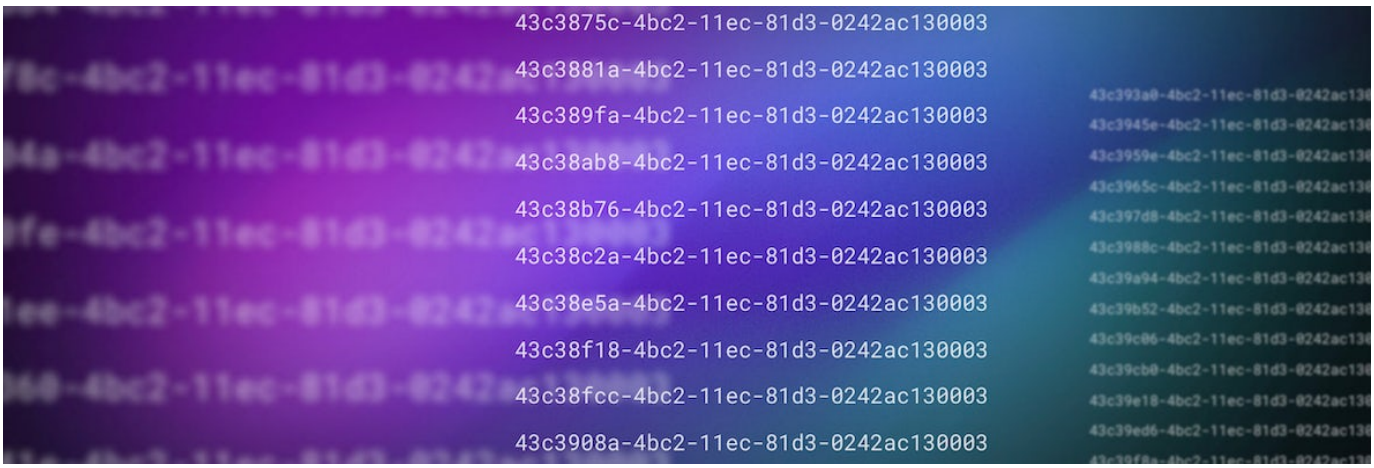
What is a UUID, and what is it used for?

Written by Charlie Custer on March 16, 2023



When working with a database, it's common practice to use some kind of `id` field to provide a unique identifier for each row in a table.

Imagine, for example, a `customers` table. We wouldn't want to use fields such as `name` or `address` as unique identifiers because it's possible more than one customer could have the same name, or share the same address.

Instead, it's a good idea to assign each row some kind of unique identifier. One option we have is to use a UUID.

What is a UUID?

A UUID – that's short for Universally Unique IDentifier, by the way – is a 36-character alphanumeric string that can be used to identify information (such as a table row).

Here is one example of a UUID: `acde070d-8c4c-4f0d-9d8a-162843c10333`

UUIDs are widely used in part because they are highly likely to be unique *globally*, meaning that not only is our row's UUID unique in our database table, it's probably the only row with that UUID in any system *anywhere*.

(Technically, it's not *impossible* that the same UUID we generate could be used somewhere else, but with 340,282,366,920,938,463,463,374,607,431,768,211,456 different possible UUIDs out there, the chances are *very* slim).

What are UUIDs used for?

To answer this question, let's imagine we're operating an ecommerce bookshop. As orders come in, we want to assign them an id number and store them in our `orders` table using that number.

We *could* set up sequential IDs such that the first order to come in is `1`, the second is `2`, and so on, like so:

id	item	buyer	price
1	The Years of Rice and Salt	Sue	$14
2	A Darkling Sea	Al	$20
3	Too Like the Lightning	Mei	$25

And this approach might work well, at least for a while, if our scale is small. However, it has some major

downsides:

First, it can easily create confusion when we're doing things like joining tables or importing new data, because the `id` values above aren't unique. This can create problems even internally if we use the same ID system for multiple tables, but it really gets messy when we start working with any kind of outside data.

Imagine, for example, that our little bookshop grows, and we acquire another online bookshop. When we go to integrate our `order` tables, we find that they've used the same system. Now we've got two order `1`s, two order `2`s, etc., and to resolve the issue, we'll have to update *every single ID* in at least one of the two databases we're integrating. Even in a best case scenario, that's going to be a tremendous hassle.

Second, the sequential approach doesn't work well in any kind of distributed system, because it means that `INSERT` commands must be executed one by one. This restriction can cause major performance issues at scale. Even if your application requires strict ID ordering, using a feature such as CockroachDB's Change Data Capture may allow you to meet those requirements while still using UUIDs and not taking the performance hit that comes with sequentially-ordered IDs.

Other traditional approaches to unique IDs, such as generating random IDs with `SERIAL`, can also lead to hotspots in distributed systems because values generated around the same time have similar values and are thus located close to each other in the table's storage.

UUIDs solve all of these problems because:

- They're globally unique, so the chances of encountering a duplicate ID even in external data

are very, *very* small.
- They can be generated without the need to check against a central node, so in a distributed system, each node can generate UUIDs autonomously without fear of duplication or consistency issues.

Reason #1 alone is a good argument for using UUIDs in almost any database system. As a business that aspires to operate at scale, reason #2 is also very relevant to our bookshop, because distributed databases offer the best scalability and resilience.

Disadvantages of UUIDs

The only significant disadvantage of UUIDs is that they take up 128 bits in memory (and often a bit more when we include metadata). If minimizing storage space is absolutely mission-critical, clearly storing a sequential ID (which will probably range somewhere between 1-10 numeric characters) is going to be more efficient than storing a 36-character alphanumeric.

However, in most cases the disadvantages of using something like a sequential identifier significantly outweigh the minimal increase in storage costs that comes from using UUIDs. UUIDs are extremely popular and widely used for a variety of different identification purposes. We've focused on database examples in this article because we make a pretty awesome database, but UUIDs are also used in analytics systems, web and mobile applications, etc.

Examples of UUIDs

There are several different types of UUIDs:

Version 1 and version 2. Sometimes called time-based UUIDs, these IDs are generated using a combination of datetime values (reflecting the time the UUID is being